Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2018

Alexandre Daniel Moreira Oliveira

Ferramenta *web* para pesquisa em conteúdos de tabelas

Web-based tool for searching tables' contents

Universidade de Aveiro Departamento de
Electrónica, Telecomunicações e Informática,
2018

Alexandre Daniel
Moreira Oliveira

Ferramenta *web* para pesquisa em conteúdos de tabelas

Web-based tool for searching tables' contents

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Sérgio Guilherme Aleixo de Matos, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president	**Doutor Joaquim Manuel Henriques de Sousa Pinto** Professor Auxiliar, Universidade de Aveiro
vogais / examiners committee	**Doutor(a) Carla Alexandra Teixeira Lopes** Professora Auxiliar, Departamento de Engenharia Informática da Faculdade de Engenharia da Universidade do Porto
	Doutor Sérgio Guilherme Aleixo de Matos Professor Auxiliar em Regime Laboral, Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Em primeiro lugar, agradeço aos meus orientadores, Sérgio Matos e José Luís Oliveira, pelo incentivo, disponibilidade e dedicação. Sempre me ajudaram a ultrapassar os obstáculos mais difíceis e me fizeram sentir capaz.

Agradeço também a toda a minha família, especialmente aos meus pais que me apoiaram de forma incondicional, à minha avó que sempre se preocupou comigo, ao meu padrinho que sempre acreditou nas minhas capacidades, e ao meu primo Gonçalo que me proporcionou momentos de maior brincadeira.

Agradeço, em especial, à minha namorada Rita que partilhou comigo todos os momentos e me deu força sempre que precisei.

Agradeço ainda aos meus amigos e companheiros diários que me ajudaram a concretizar esta etapa e com quem vivi momentos de trabalho mas também de descontração.

A todos aqueles que me tiveram no pensamento durante esta fase da minha vida.

Palavras-chave Mineração de Texto, Mineração de Tabelas, Reconhecimento de Conceitos, Recuperação de Informação, Bioinformática.

Resumo

O número de artigos biomédicos está constantemente a crescer e os investigadores têm cada vez mais dificuldade em encontrar informação relevante, comparar resultados e identificar novas hipóteses de forma eficiente. As técnicas de mineração de texto têm sido exploradas para desenvolver sistemas que forneçam acesso fácil e rápido à literatura científica. O problema é que muitas destas ferramentas ignoram completamente as tabelas e apenas processam as partes textuais.

Esta dissertação foca-se na análise e indexação de tabelas extraídas de artigos científicos, dado que muitas vezes estas incluem bastante informação que pode ser útil para os investigadores e não está disponível no restante conteúdo das publicações. Assim, o principal objetivo deste trabalho é criar uma estrutura de indexação flexível capaz de lidar com diferentes formatos de tabelas e identificar conceitos biomédicos referidos nas próprias tabelas, nas legendas e no texto que referencia as tabelas.

Foi então desenvolvida uma ferramenta *web* que permite aos utilizadores pesquisar e visualizar tabelas anotadas extraídas de artigos científicos. A solução encontrada usa algumas ferramentas de código aberto, nomeadamente o Neji para o reconhecimento de conceitos e o Elasticsearch para a indexação de texto.

Keywords Text Mining, Table Mining, Concept Recognition, Information Retrieval, Bioinformatics.

Abstract

The number of biomedical articles is constantly growing and researchers have more and more difficulty to efficiently find relevant information, compare results and identify new hypotheses. Text mining techniques have been explored to develop systems with the aim of providing easy and fast access to scientific literature. The problem is that most of these tools completely ignore tables and just process textual parts.

This dissertation focuses on the analysis and indexing of tables extracted from scientific articles, as they often include a lot of information that can be useful to researchers and it is not available in the remaining content of the publications. So, the main objective of the work is to create a flexible indexing structure to handle different table formats and recognize biomedical concepts referred in the tables themselves, their captions and texts that reference them.

A web-based tool was developed to allow users to search and visualize annotated tables extracted from scientific articles. The solution found uses some open-source frameworks, namely Neji for concept recognition and Elasticsearch for text indexing.

Contents

List of Figures

List of Tables

Listings

Acronyms

TM Text Mining

NLP Natural-language Processing

POS Part-of-Speech

IR Information Retrieval

IE Information Extraction

NER Named Entity Recognition

UMLS The Unified Medical Language System

HTML Hypertext Markup Language

PDF Portable Document Format

SVM Support Vector Machine

CSS Cascading Style Sheets

XML Extensible Markup Language

CRF Conditional Random Fields

JSON JavaScript Object Notation

DSL Domain Specific Language

NRT Near Real Time

CSV Comma Separated Values

TSV Tab-separeted Values

PMC PubMed Central

Chapter 1

Introduction

The need to keep records about everything encouraged human beings to devise ways to store and share information. In a recent past, prints were the standard manner to store data, but nowadays, paper is gradually giving way to digital storage. Subsequently, data volumes are exploding and it is important to develop tools that can provide easy and efficient access as well as emphasize relevant knowledge derived from data.

Biomedicine is no exception to this tendency, scientific articles are constantly produced and published to report results and findings. Statistics about MEDLINE, an online database of life sciences and biomedical information, sustain this fact, since it contains over 23 million citations of biomedical literature and the number of added citations per year is growing [1], as Figure 1.1 illustrates.

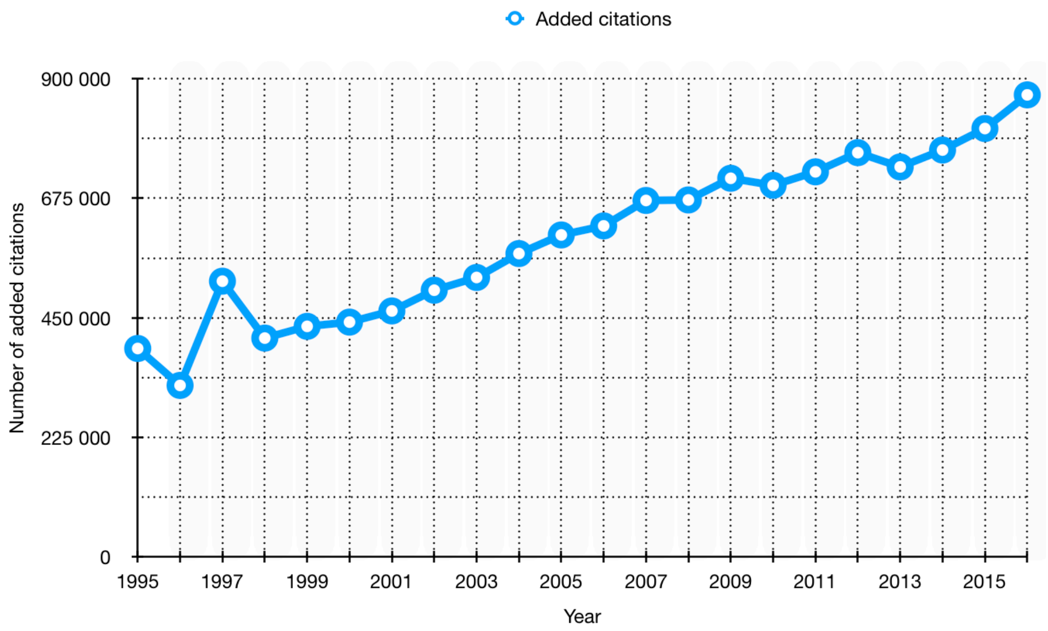

Figure 1.1: Added citations to MEDLINE per year (based on [1])

Scientists can not manage this quantity of articles by themselves and it is very important to develop tools that can help them finding information about topics of their own interest, and at the same time to provide extra knowledge extracted from the same sources. As a result, machine learning and text mining techniques have been applied to this field to overcome several challenges associated with information overload and consequently to create useful systems to biomedical researchers [2].

Existent systems usually focus on raw text parts, ignoring tabular data. However, tables are important ubiquitous elements that are widely used by researchers to compact relational data containing meaningful information that can be only available there [3]. Therefore, it is interesting to explore this type of data and make the most of it, by extracting information from it and being able to retrieve it. Nevertheless, tables should not be totally isolated because their surrounding text gives some context and usually explains their contents [4].

The aim of this work is to study information retrieval and information extraction topics in order to develop a system that applies these techniques on tables from biomedical publications and to create a web-based tool to retrieve and visualize tables as well as information extracted from them. To accomplish that, it was necessary to understand the structure of articles and of their tables to be able to design a solution capable of handling different types of tables. Captions and text that reference tables were also considered and used to complement table's data. Named entity recognition was performed to find biomedical concepts, and all data was stored in an index that can be queried.

This resulted in a web-based tool capable of retrieving annotated tables as well as their captions, notes, text with references to them and links to their original articles. Annotations are marked by type and text is mostly formatted according to the original sources.

Chapter 2 studies information retrieval and information extraction challenges and their use in biomedical literature and tables. Some open source text indexers and annotators are discussed in order to choose the proper ones to include in a practical solution. **Chapter 3** examines the dataset used and how tables are stored as well as their related information. It also presents the implemented system, describes its main modules and exposes some faced challenges. **Chapter 4** discusses some results obtained by testing the system and presents the web interface. Finally, **Chapter 5** gives some final remarks and suggests topics to be considered in further work.

Chapter 2

State of the art

The advances of computer science provided new ways to store, search and share information. Consequently, new scientific knowledge is continually generated and reported through the scientific literature producing unmanageable quantities of data. In particular, the amount of biomedical publications are expanding at an increasing rate. Statistics about Medline show that the database contains over 23 million citations of biomedical literature and is growing at the rate of 800 thousand new citations per year [1].

For researchers, it has been increasingly difficult to satisfy their information needs and to be updated with all the emerging research findings. As a result, text mining methods have been used to make biological literature more accessible and useful [5], providing novel ways that can help to deal with colossal amounts of information [2].

2.1 Text mining biomedical literature

Text Mining (TM) involves the text processing of a large collection of documents to extract relevant information, such as trends and patterns implicitly present in the corpus that can be further analyzed [5, 6]. TM tools are becoming essential for researchers because with the extracted information, they can provide an easier access to literature and also form new hypotheses to be further tested, explored and even proved by area experts using the appropriate means of experimentation [5, 7].

As biomedical papers and reports are written by humans, the development of TM tools requires the use of Natural-language Processing (NLP) techniques to analyze and represent natural language texts [8]. Such techniques involve the processing of the original text to perform transformations and linguistic analysis that usually accomplish some of the next tasks:

- **Sentence splitting**: to find sentence boundaries in order to split a text into sentences.

- **Tokenization**: to split a text into a set of tokens (words), usually removing punctuation and special characters.

- **Stopping**: to discard non-informative words with high occurrence frequencies like determiners and prepositions.

- **Part-of-Speech (POS) tagging**: to assign a descriptor (tag) to each token that represents its grammatical category (e.g., noun or verb).

- **Term normalization**: convert all different tokens that represent the same term to a unique common token (e.g., *USA* and *U.S.A.* become *USA*).

- **Stemming**: to remove affixes of inflectional forms and derivational words keeping a representation of the main essence of the word (e.g., *hypertension* and *hypertensive* become *hypertens*).

- **Lemmatization**: to reduce inflectional forms and derivational words to their corresponding base forms (e.g., *am*, *is* and *are* become *be*).

These pre-processing tasks are used to prepare and simplify text inputs that will be submitted to other text mining techniques in order to obtain the desired information. Thus, it is usual to focus on two big topics that are related to text mining:

- **Information Retrieval (IR)**: "finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)", defined by Manning et al. [9];

- **Information Extraction (IE)**: "extraction of structured information such as entities, relationships between entities, and attributes describing entities from unstructured sources", defined by Sarawagi [10].

For example, considering the biomedical field, IR can aid with the identification of relevant papers about topics of the researcher's interest (e.g., heart attack) and IE can pull out specific facts from documents that possibly would be imperceptible to the researcher (e.g., hypertension increases the risk of heart attacks) [11]. In other words, IR returns a set of relevant documents whereas IE returns extracted facts from them [12].

2.1.1 Information Retrieval

Information Retrieval systems can actually provide tools that users can exploit to make fast and accurate searches usually based on a set of keywords related to a desired topic. Nowadays, many human beings depend on them to find information that can aid their work, business, education, and entertainment. With the advances of IR techniques, this kind of systems are becoming increasingly more accurate and robust. For example, the main objective of Web search engines, probably the most used type of IR systems today, is to find web pages. However, search engines, such as Google, can actually do more than retrieve web pages, they can also facilitate people finding information about movies, celebrities, organizations and events, comparing products prices and even answering to questions [13].

Figure 2.1 shows a generic architecture of an Information Retrieval system. A search is always triggered by an information need, the desire to find a specific information about a certain topic in order to satisfy a user's lack of information. Thus, user formulates a query, a set of words that in some way represents his information need. This query is forwarded to a search engine, an implementation of IR techniques that executes two basic operations:

- **Indexing**: to store documents in a structure that allows efficient searches;
- **Search**: to retrieve indexed documents based on a query.

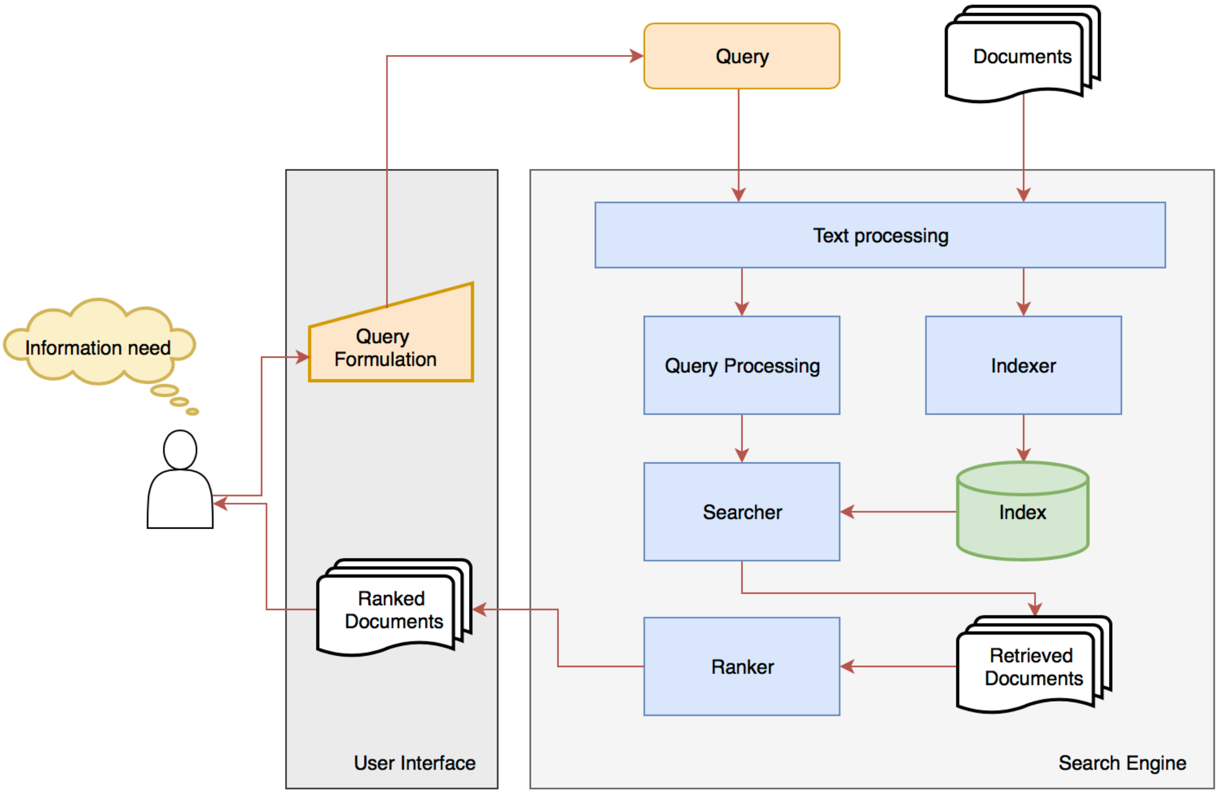

Figure 2.1: Generic architecture of an IR system (based on [14])

The **indexing** task starts with the input documents that are submitted to some text processing like tokenization, stemming and stopping. Indexer receives these documents and constructs a structure (index) that stores information about them allowing fast and accurate searches. The inverted index is the most common structure to build this component. Its basic idea is to construct a dictionary with the terms that occur in the indexed documents, called vocabulary, and map each term with a list of postings. A posting is an item that associates a document with a term that occurs in it [9].

A **search** depends on a query that is also submitted to some text processing just like the input documents. After that, some additional operations may be performed to obtain better results (e.g., query expansion). The processed query is forwarded to the searcher that retrieves a list of relevant documents based on the index. Finally, it is usual to use a ranker that sorts retrieved documents by relevance in order to have documents more related to the query on top of the list.

2.1.2 Information Extraction

While information retrieval has a very concrete objective of finding relevant documents, information extraction is related with the extraction of many kinds of semantic content. This implies the existence of several sub-tasks with different but related goals [15]. The main ones are:

- **Named Entity Recognition (NER)**: to identify entities referred in text and classify them according to their meaning (e.g., chemicals, proteins, diseases);
- **Co-reference resolution**: to find different text expressions that refers to the same entity;
- **Relation extraction**: to extract semantic relations between entities;
- **Summarization**: to extract some parts of the original text that define the main themes or ideas that are addressed.

Named entity recognition will be focused in this dissertation since it is one of the most important tasks in information extraction because other tasks, like co-reference resolution and relation extraction, usually require entities as input. Thus, it must be performed first and should be accurate, since it will influence the results of the other tasks.

Figure 2.2 illustrates the generic architecture of a NER system. It is composed by three main modules: **text pre-processing**, **entities recognition** and **post-processing**. In biomedical literature, it is particularly difficult to apply NER techniques because of the specific vocabulary and nomenclatures used in the domain (e.g. the same abbreviation can represent distinct entities). Therefore, in general, biomedical NER systems follow this architecture but each module has to be very well adapted and optimized for the biomedical language [16].

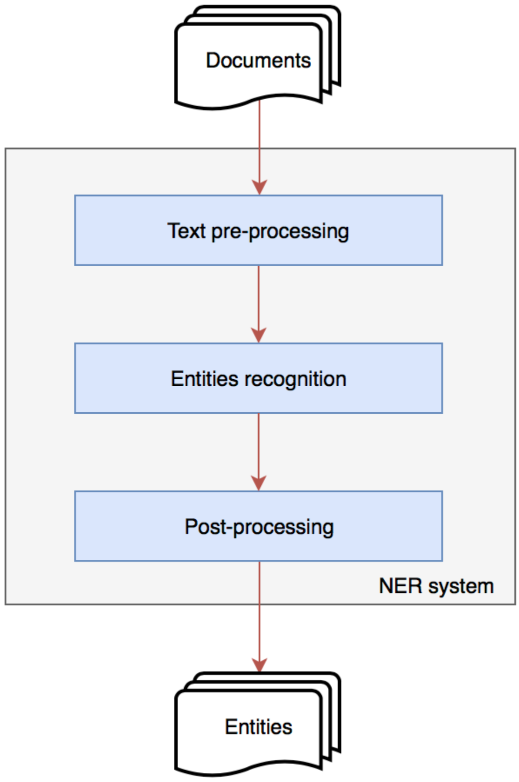

Figure 2.2: Generic architecture of a NER system (based on [16])

Text pre-processing module

Text pre-processing module performs natural language processing operations like tokenization, sentence splitting and stopping. Here, tokenization is indispensable, since the entity recognition itself is performed based on tokens. The quality of the tokens will hardly influence annotations. As biomedical language is very complex, the tokenizer as well as all other NLP operations used must be suitable and optimized for this specific domain.

SPECIALIST NLP [17] and GDep [18] are some available tools for biomedical text processing. LingPipe [19] and OpenNLP [20] are more generic but can also be adapted to the domain by training models. These last two even have modules to perform entity recognition.

Entities recognition module

The most common methods used to recognize entities are based on rules, dictionaries or machine learning. Hybrid approaches, that combine some of the previous ones, are common too.

Ruled-based approaches find entities based on a set of hand-made rules that explore syntactic and orthographic patterns (e.g. regular expressions). It can only be applied to detect entities that have a very concrete common structure (e.g. proteins [21]).

Dictionary-based approaches work by matching biomedical names stored in a curated collection of dictionaries with the input text. Each entry of a dictionary usually contains a set of distinct expressions referring to the same concept and an identifier code that classifies that concept according to a knowledge base. An example knowledge base is The Unified Medical Language System (UMLS) that tries to establish a standard in biomedical vocabulary containing about 12 million of distinct concept names [22].

Machine-learning approaches are based on pre-trained models that predict the existence of entities in the text. Thus, it requires proper annotated datasets to train generic models, like Neural Networks or Support Vector Machines. An additional concern is the extraction and selection of proper features from tokens that will be used as input to the model.

Post-processing module

The last module aims to improve annotations by removing mismatched entities or disambiguating entities that were classified with multiple identifiers. Additional entities can be detected in this phase, usually spelling variants of entities already annotated, like abbreviations.

2.2 Table mining

Most of the text mining works on biomedical literature focus on text parts and do not often consider table data. However, tables are widely used by biomedical researchers because of their easiness to compact relational data, such as experimental results, from which is possible to extract precious information.

To take advantage of this type of data, it is interesting to use text mining methods on tabular information. So, table mining is the application of TM techniques to retrieve and extract information from tables. However, tables are not plain text and consequently it is not so direct to mine them, and it is necessary to overcome some obstacles:

- There are multiple ways to digitally represent tables and the existing formats are usually visually oriented. Because of that, tables are not easy to process since each format needs a specific approach;

- Tables are designed to be read by humans and because of their flexibility, they can assume several distinct structures. There is not a common way to read any possible table and understand the meaning of its information.

Considering these problems, three main sub-tasks can be accomplished to extract information from tables [23, 24]:

- **Detection**: to locate a table and be capable to read its cells within a document.

- **Functional and structure analysis**: to distinguish the function of each table cell (header or value) and associate each value to its correspondent header.

- **Interpretation**: to extract relationships between cells.

2.2.1 Table detection

Table detection process depends on the ability to locate a table and distinguish its components within a specific file allowing data extraction from the table. Most of the efforts in table detection are focused on Hypertext Markup Language (HTML) and Portable Document Format (PDF) files.

HTML

At first glance, it seems to be easy to extract tables from HTML files because there are specific tags designed to handle tabular information [25]. Tables are identified by the **<table>** tag, so an easy way to detect them relies on the ability to find that tag occurrences. However, this tag is also used to format layouts of other page elements like forms and menus with a multi-column layout making the detection phase harder [26, 27].

Chen et al. [27] proposed to filter HTML tables based on heuristic rules. They do not consider that a **<table>** tag represents an actual table if it contains only one cell or if its content contains a considerable number of hyperlinks, forms and figures. Beyond this, they measured the similarity between neighbour cells based on comparing its strings, entities and number of characters. Only tables presenting a similarity between cells above a threshold are considered as real tables. The combination of these rules achieved 86.5% F-measure.

Babatunde et al. [28] provided a Hidden Markov Model approach. To include other file formats, Word and PDF documents were converted to HTML and tables were also detected using the same method. The used dataset was self-generated and composed by 526 tables stored in 25 documents of the three referred formats. Considering the entire dataset, tests showed an F-measure of 87.8 %.

Wang and Hu [26, 29] tried several machine learning techniques like Naive Bayes, K-Nearest Neighbors and Support Vector Machine (SVM). The best F-measure achieved was 95.89% applying SVM. They noticed that a table was wrongly classified because **<p>** tags were used to format its rows.

Lerman et al. [30] also said that looking for tables by tags were not a good approach. Besides the usage of table tags to format layouts, in a large fraction of HTML documents table data are formatted with other tags instead of the standard ones.

In addition, Krupl and Herzog [31] mentioned that tables can even be formatted with Cascading Style Sheets (CSS) or Javascript. To handle it, they presented a different approach that relies on rendering web pages using Mozilla browser instead of process HTML. By image analysis they detected tables according to a set of heuristics. The algorithm could detect 70% of the total number of tables but authors reported that the process is very slow because Mozilla takes some time to render pages.

Note that table detection in Extensible Markup Language (XML) files can be similarly performed as in HTML, that is by finding tags. The advantage is that the difficulties related with using **<table>** tags for formatting purposes are not present in XML. However, tags used in XML depend on the publisher's choice and consequently they could not be the standard ones that are used in HTML. In general, it is easier to extract tables from XML files but it is harder to develop generic solutions, since tables are usually stored differently.

PDF

Yildiz et al. [32] proposed a system called *pdf2table* that detects PDF tables and its cells based on heuristics and produces an XML file that can be further processed. It uses a tool [33] that processes PDF files and returns an XML with all text elements and their coordinates in the original file. It assumes that tables have more than one column and merges lines of text to form blocks containing a possible table. They used two separated datasets, one with simple tables and another with more complex tables. For the complex dataset, they achieved an F-measure of 93.48% on table detection and 81.99% on decomposing tables in cells.

Similarly, *PDF-TREX* [34] detects PDF tables based on heuristics producing an XML representation of its cells. It considers a PDF document as a Cartesian plane and uses the positions of words to detect tabular arrangements exploiting the spatial distribution of page elements. Tests showed an F-measure of 91.97% on table detection and 84.61% detecting cells.

Liu et al. [35] used the concept of sparse lines, that is, lines which the minimum space between their consecutive words is above a threshold or lines with a small number of characters. These lines were classified as rows of tables or not through Support Vector Machine and Conditional Random Fields (CRF) and afterwards they were iteratively merged to form a table. The best F-measure achieved was 96.36% using the CRF version.

2.3 Frameworks

There are several frameworks that can be part of an information retrieval and information extraction system for tables. Three types of frameworks can be considered: web frameworks, search engines and annotators.

Web frameworks will not be focused because any of the most popular ones (e.g. Django or Spring) is perfectly viable to fulfill the requirements. The targets will be annotators and search engines, to perform named entity recognition and to index/retrieve all pertinent information. Thus, multiple open source solutions are discussed in this section in order to choose one of each type to be integrated in a final solution.

2.3.1 Search engines

Search engines are systems that store text, in a data structure called index, in manner that it is possible to efficiently search that index to retrieve the pretended information. The most popular search engines are Elasticsearch [36] and Apache Solr [37].

Elasticsearch

Elasticsearch [36] is an open-source distributed search engine that can store big volumes of text data and perform different types of searches. It was created by Shay Banon in 2010 and after that it quickly gained acknowledgment and became one of the most popular search engines [38, 39], being used by several well-known platforms like GitHub, Netflix and Facebook [40].

It is built on top of Apache Lucene, a full-text search engine library written in Java. This library is very complex and requires some mastery on information retrieval field to truly understand how it works and then how to use it. With that in mind, Elasticsearch uses and manages Lucene's capabilities internally to get the most out of it. Consequently, it hides some complexities of Lucene, making indexing and searching operations more abstract and simple [41].

At the practical level, it is important to understand the basics about indexing and search operations. Both operations are triggered by commands that are sent through a REST API.

With regard to indexing task, documents are expressed in JavaScript Object Notation (JSON), a set of key/value pairs. There are several types for values (e.g. string, numeric, array, nested...) and by conjugating all them it is possible to create many different document structures according with a practical problem. For each index, it is recommended to define a mapping, that is, to define some rules that indexed documents may obey [42].

Searches are assure by a flexible query language based on JSON, the query Domain Specific Language (DSL). Basically, this language works based on two types of clauses [43]:

- **leaf**: used to find a value in a specific field (e.g. term - matches an exact value on a field);
- **compound**: used to wrap a set of other clauses and combine them in a logic manner (e.g. bool - combines operations like must, should, must_not) or change their results (e.g. constant_score - sets a query score to a fixed value).

To conclude, Elasticsearch is a very complete search engine that provides multiple features. Some of them are summarized above:

- It is **distributed**, a single index can be subdivided into multiple shards that are fully-functional and independent "indexes" that can be hosted on any node. By using shards, it is possible to distribute data across several nodes and parallelize operations increasing performance;

- It is **scalable**, on one hand, it is capable to run on a single laptop or to be distributed among several servers. On the other hand, indexes can easily grow in size, storing large quantities of documents;

- It guarantees **hight availability**, because indexes can be replicated one or more times. By doing this, replica shards are created by copying primary (original) shards and allocated in a different node. Replicas can also be used to increase throughput since queries can be executed on all replicas in parallel;

- Communication is assured through a **REST API**, so it is compatible with several programming languages;

- Documents are expressed in **JSON** and supported structures are very ample and flexible. With that, several options are available to match a need of a problem, regarding storage;

- Searches are made through the **query DSL** that assures rich, robust, complex and efficient queries;

- It is a **Near Real Time (NRT)** platform, it has mechanisms to minimize the latency between the time a document is indexed and the time it is searchable (usually one second).

Apache Solr

Apache Solr [37] is an open source distributed search engine developed by The Apache Software Foundation. It is very trusted by the community, being used by several well-know companies such as Adobe, AT&T and eBay.

Solr core is very similar to Elasticsearch. It is built on top of Lucene library for the same reason: to provide full-text search capabilities through a simple interface that hides the complexity of Lucene. Beyond this, it has multiple features in common with Elasticsearch, some of them:

- It is **distributed** and **fault tolerant**, supporting multiple shards distributed among different nodes as well as replicas to guarantee **high availability**;

- It uses **open standards**, communication is assured by HTTP requests and documents are expressed in JSON, XML or Comma Separated Values (CSV);

- It is **Near Real Time**, documents are indexed and shortly after it they are ready to be searched;

Table 2.1 compares Elasticsearch with Apache Solr. Both frameworks are very complete and the main features are identical. Apache Solr supports more document formats but it is not critical because JSON provides the same flexibility than XML and even more than CSV. Text indexing capabilities are very similar, since Lucene library is used in both solutions.

	Elasticsearch	**Apache Solr**
Search Engine library	Lucene	Lucene
Communication	REST	REST
Document Formats	JSON	JSON, XML,CSV
Distributed	Yes	Yes
Near Real Time	Yes	Yes
Results highlighting	Yes	Yes
Query suggester	Yes	Yes

Table 2.1: Search engines comparison

2.3.2 Annotators

Annotators are systems that perform named entity recognition producing annotated files. There are not specific frameworks to annotate tables, so they must be transformed into just text that will be processed by such tools. Some available annotators for biomedical terms are MetaMap [44], ConceptMapper [45], BANNER [46] and Neji [47].

MetaMap

MetaMap [44] is a ruled-based tool that recognizes biomedical concepts on text and maps them to UMLS terms. It detects candidate entities by performing several NLP operations and comparing chunks of text with strings derived from the UMLS Methathesaurus, a thesaurus organized by concept. Final results are determined based on a score calculated from four measures: centrality, variation, coverage, and cohesiveness. These measures are related with linguistic characteristics and how well chunks of text matched UMLS strings.

It is highly configurable, since it allows users to set option flags that control all distinct modules. The output is configurable too, including XML, JSON and human-readable formats.

ConceptMapper

ConceptMapper [45] is an open source tool that recognizes entities based on dictionaries. These dictionaries are stored in XML and contain entities represented by multiple variants (synonyms). Entities' meta-data is stored as attributes that can be set to each entity (e.g. POS function). Variants inherits their parent (entity) attributes and it is possible to override them. A custom tokenizer is used for text input and terms from dictionaries, since entities are recognized by exact matching.

Like MetaMap, it is highly configurable, allowing users to turn on/off NLP operations and to choose different strategies to be followed while matching tokens with entities from dictionaries.

BANNER

BANNER [46] is an open-source biomedical NER system implemented in Java. The recognition is performed based on a machine learning technique, since it uses CRF models implemented by Mallet [48]. It uses a custom tokenizer that produces three types of tokens: words, numbers or punctuation marks. Part of speech tagging and lemmatization are performed by Dragon toolkit [49]. Disambiguation is not considered by this tool. It was tested recognizing genes and diseases/treatments and achieved F-measures of 81.96% and 54.84% respectively.

Neji

Neji [47] is a framework for concept recognition in biomedical texts that works based on a processing pipeline, a sequential execution of several modules. It performs NER through dictionary matching and machine learning methods. Figure 2.3 shows the processing pipeline and its corresponding modules. Each module focus on the execution of a specific operation and its output is forwarded to the next one.

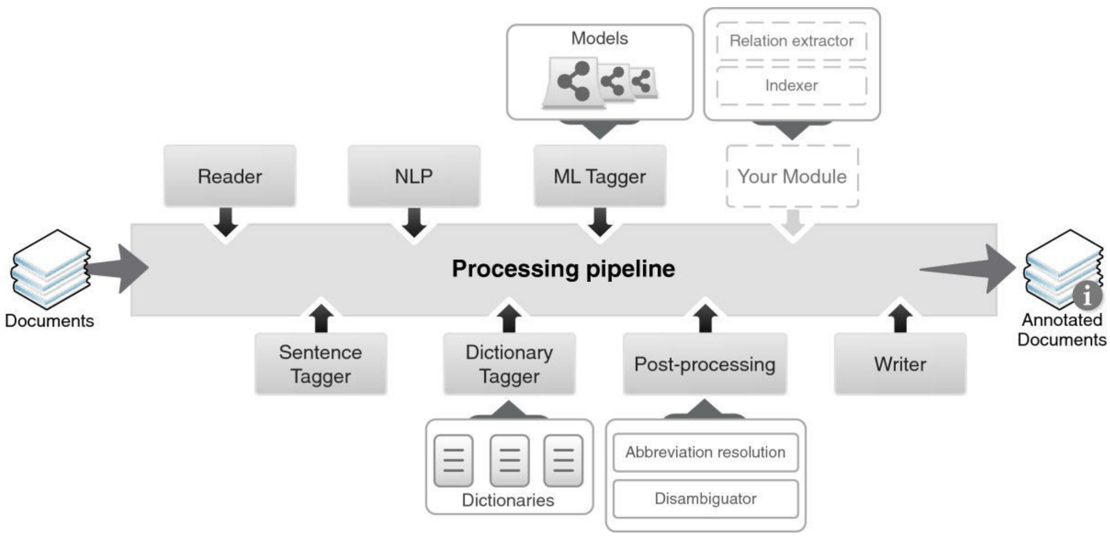

Figure 2.3: Processing pipeline and modular architecture of Neji [47]

The **Reader** module reads the input and collects relevant data that will be processed by the following modules. It supports XML or plain text as input. When using XML, it is possible to specify a set of tags to be considered in the process.

All operations are based on sentences, so the **Sentence Tagger** module uses LingPipe [19] library to perform sentence splitting .

The **NLP** module uses GDep [18] to perform tokenization, part-of-speech tagging, lemmatization, chunking and dependency parsing.

The **Dictionary Tagger** module matches terms with dictionaries that are stored in Tab-separeted Values (TSV) format. It supports multiple dictionaries that can be configured with a priority level that is used later in case of ambiguity. The comparisons are not case sensitive and only exact matches are considered. Terms with less than three characters and non-informative words are discarded.

The **ML Tagger** module uses several CRF models to recognize distinct types of entities (e.g. genes and proteins). Although, normalization is not supported by these models, so it uses an algorithm based on dictionaries to normalize results.

The **Post-processing** module detects abbreviated forms of previous recognized concepts and matches them to the same identifier. Beyond this, it can remove some annotated concepts based on rules and disambiguate annotations.

Finally, the **Writer** module stores results to be further processed. Several formats are available (e.g. JSON) and it can produce more than one format at the same time.

Table 2.2 compares strategies used by these four annotators. Neji is the most complete of them. It can combine dictionary matching with machine learning models to perform the recognition. Furthermore, it is the most configurable tool, custom dictionaries can be easily added and another models can also be trained to predict new types of entities. Input and output formats as well as NLP operations can be configured too.

		MetaMap	ConceptMapper	BANNER	Neji
Text pre-processing	Tokenization	X	X	X	X
	POS tagging	X	X	X	X
	Stemming	X	X	X	X
	Stopping		X		X
Entities recognition	Rules	X			
	Dictionaries		X		X
	Machine Learning			X	X
Post-processing	Disambiguation	X			X

Table 2.2: Annotators comparison

Chapter 3

Implementation

This chapter focuses on the implemented system and describes its main modules. Some faced challenges are presented as well as the decisions made to overcome them.

3.1 System Overview

In practical terms, the main objective of this dissertation is to build an information retrieval and information extraction system focused on tables from biomedical publications. To develop such a system, it is necessary to successfully accomplish some requirements:

- Iterate over biomedical articles in order to locate tables and related information;
- Create a flexible data model that temporally stores read information;
- Perform information extraction operations over the data model;
- Index original table data and the additional information obtained by information extraction procedures;
- Develop a web interface to query indexed data and visualize results.

In other words, the whole process requires some data transformation. Original documents containing tables may be transformed into a searchable structure that stores data contained in the original tables and the extracted information. With that in mind, a system was developed whose architecture is illustrated by Figure 3.1. It is composed by four distinct components which functions can be roughly described as:

- **PMC Parser**: reads PubMed Central (PMC) articles to extract tables and some related data (caption, foot and body paragraphs with a reference to a table) and stores it internally;
- **Neji**: finds biomedical concepts all through the data stored by PMC Parser;

- **Elasticsearch**: indexes table's data and concepts found by Neji and allows efficient searches;

- **Web Interface**: provides a user interface that allows querying operations on the Elasticsearch index.

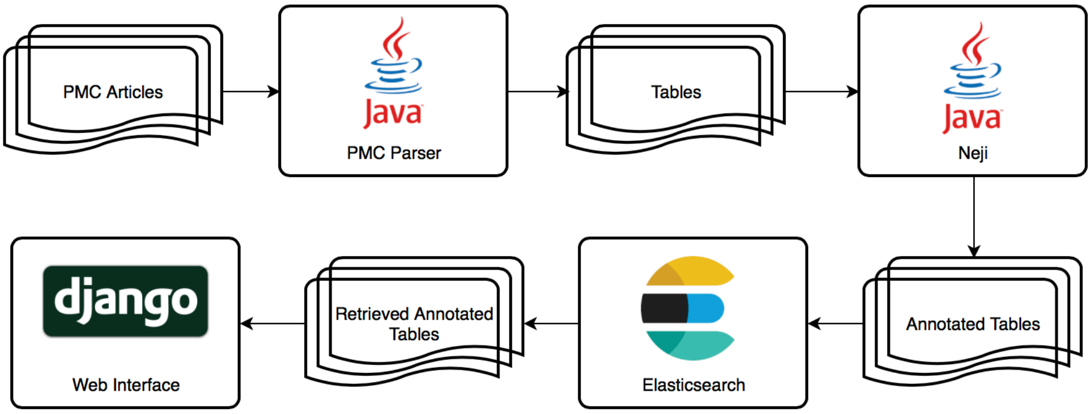

Figure 3.1: System architecture

Next sections describe the dataset used as well as PMC Parser, Neji and Elasticsearch modules. The Web interface is presented in Chapter 4.

3.2 Dataset

PubMed Central (PMC) is a free electronic repository of biomedical and life sciences journal articles containing over 4.8 million publications. Although it provides free access to its contents, only a part of the collection is distributed under a more liberal license that allows automated operations. This fraction is called PMC Open Access Subset and contains almost 2 million articles available as PDF, XML and TXT files [50]. It was decided to use XML ones because they are easier to process in terms of table detection and functional analysis. As stated in Section 2.2, tables stored in XML files are easy to identify because a dedicated tag is usually used to delimit tables. Header cells are identified too, facilitating the function analysis. Note that XML files do not share the main problem found in HTML ones, namely the use of *table* tags to format web pages.

To create a specific reader for this dataset it is necessary to understand the structure of these files. Thus, some of them were randomly picked and manually analyzed by comparing the source XML with the correspondent article available online. Although it was more focused on tables, all content was inspected.

At the top of each file there is a header, delimited by a *front* tag, containing some meta data about the article, including its journal, publisher, title, PMC identification, authors, publication date and abstract text.

Then, there is a *body* tag that contains the article itself. It is divided in sections (*sec* tag) that are composed by a title (*title* tag) and a set of text paragraphs (*p* tag). Additionally, sections may also contain more complex structures, namely figures (*fig* tag) and tables (*table-wrap* tag). Figures were not analyzed because they are out of scope. All text fields (e.g. paragraphs or cells) may also contain another tags indicating some text formatting, like *italic* and **bold**. Formulas (in LaTeX and Mathematical Markup Language) and hyperlinks are regular too, they are marked by *inline-formula* and *ext-link* tags.

As mentioned, tables are defined by a *table-wrap* tag. Inside it, obviously there is a table and some related information that may be helpful. Figure 3.2 compares an example table obtained from the PMC online repository and its correspondent representation in XML (from PMC4255783).

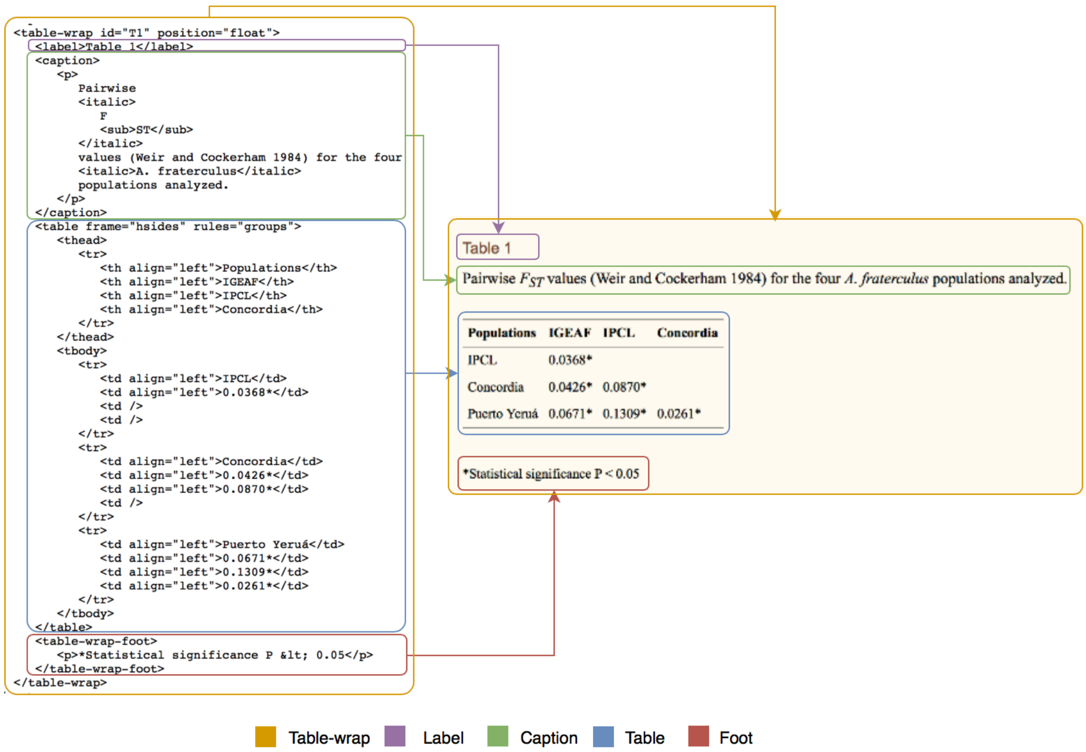

Figure 3.2: Example *table-wrap* and its visual representation (from PMC4255783)

As it is visible, *table-wrap* tag contains four other tags delimiting the next fields:

- ***label***: expression that identifies the table, usually the word "table" plus the number of appearance in the article;
- ***caption***: a paragraph containing a small portion of text that usually describes what is the table about;
- ***table***: the table itself, stored using HTML syntax;

- ***table-wrap-foot***: additional text, usually some notes, that complements the table and explains some details.

Note that *table-wrap* tag has an attribute named *id* that identifies the table inside the article. While *label* is more user-oriented, this *id* is mostly used to make references to tables, so it will be very important to the system. It can be also used to create a unique identifier for each table of the dataset, no matter which article, by combining it with the article's PMC *id*.

Some fields, like *caption* and *foot*, just contain text. In these cases, it is only required to be alert that there are tags that represent some kind of text formatting or formulas mixed with the real text. In contrast, tables are much more complex and need special attention. Fortunately, PMC dataset uses a very well known set of tags to define tables: the HTML ones. This simplifies a lot the task of reading tables. Table 3.1 resumes main HTML tags used to represent tables in this dataset.

Tag	Description	Parent tag
thead	Encapsulates header rows	*table*
tbody	Encapsulates data rows	*table*
tr	Encapsulates a row	*thead* or *tbody*
th	Defines a header cell	*tr*
td	Defines a data cell	*tr*

Table 3.1: HTML tags used to represent tables in XML articles from PMC

It is standard to use *th* tags for header cells and *td* tags for data cell. Although, the dataset contains some tables where the header is composed by *td* tags instead of *th* ones. This small variation needs to be considered by the PMC reader.

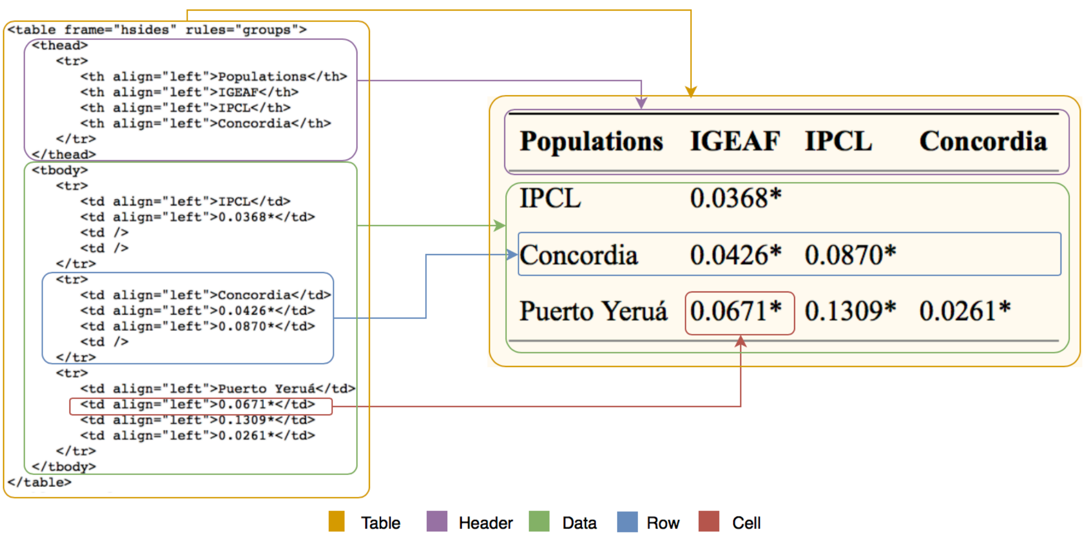

Figure 3.3: Example table and its visual representation (from PMC4255783)

Figure 3.3 has an example table that shows the main tags used to represent tables in the dataset and their correspondent visual representation. It is possible to see that *align* attribute is used in some *td* and *th* tags. Although this attribute does not give any crucial information, *rowspan* and *colspan* attributes are used very often and have a lot of influence in table's structure. Ocasionally, the content of some cells is just an *hr* tag, representing a line.

It was noticed that some tables have a very peculiar characteristic, they are composed by multiple independent tables, as showed in Figure 3.4. When this happens, multiple *table* tags, representing the smaller tables, can be found inside a unique *table-wrap* tag with the same label, caption and foot.

Table 2.

Acceptance Criteria

Parameter	Acceptance criterion (PPC well level)
R_{SEAL} (Baseline)[a]	>100 MΩ
Current amplitude (Baseline)	>0.2 nA
R_{SEAL} stability (between first and second additions)	<50% decrease

	Acceptance criterion (PPC plate level)
Z′ factor	≥0.5
Success rate (% valid wells)	>90% accepted wells per PPC plate
EC/IC_{50} for reference compounds	≤0.5 log from historical mean

[a]Typical R_{seal} values ranged between 200 and 1,000 MΩ.

PPC, population patch clamp.

Figure 3.4: Example table composed by multiple other tables (from PMC4991607)

3.3 PMC Parser

Considering the structure of the articles described in Section 3.2, a Java program was created with aim of extracting tables from these articles and storing them internally in a manner that allows some information extraction operations and also their posterior indexing. To achieve that, it is necessary to design a data model and build a reader capable of doing such transformation.

3.3.1 Data model

A data model was designed in order to store tables, including all important fields. Figure 3.5 shows a class diagram that represents it.

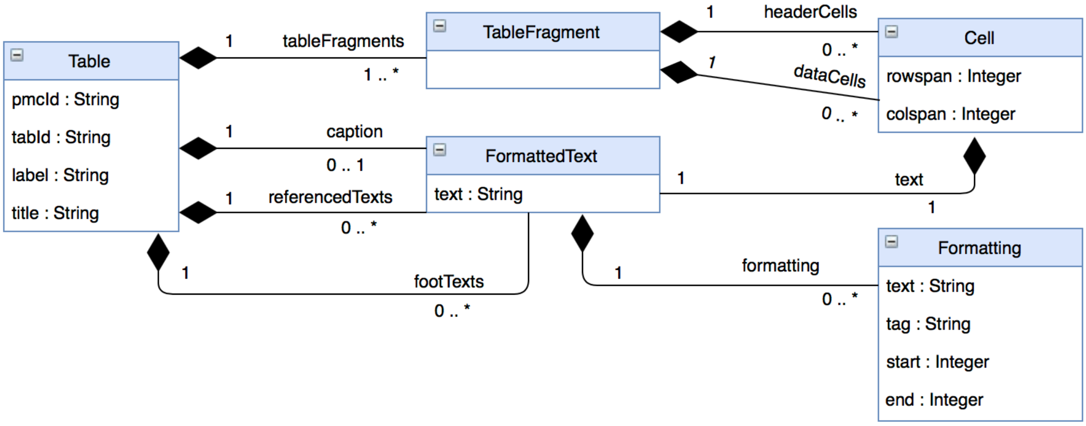

Figure 3.5: Data model class diagram

The idea here is to create one *Table* object for each *table-wrap* found. All its fields were considered and the multiple *table* tags problem was also fixed by including a list of *TableFragment* objects for each *Table* object. Text formatting was also included to allow a better visualization when reproducing results. Below, all these classes are described in detail.

Formatting class

A *Formatting* object represents a text modification, like *italic* or **bold**, hyperlinks and formulas with regard to a specific portion of text. To be able to know where a formatting takes place, the character positions where it starts and ends are stored. These positions are always regarding to the *text* of a *FormattedText* object. A *tag* is stored to identify the formatting type. A *text* field is included to store additional information. It is only used to store the url when the formatting is an hyperlink, or to store the text representation of formulas in LaTeX or Mathematical Markup Language.

Fields	Type	Description
text	String	Stores additional information. It only has information when the object represents a formula or an hyperlink
tag	String	Tag that identifies a formatting type (e.g. *italic* or **bold**)
start	Integer	Character position where the formatting starts, with regard to a *FormatedText* object.
end	Integer	Character position where the formatting ends, with regard to a *FormatedText* object.

Table 3.2: Fields' description of *Formatting* class

FormattedText class

FormattedText class represents a piece of formatted text. It contains a set of style modifications that occurs on it. All text extracted from articles will be stored as a *FormattedText* object, excepting ids, labels and titles.

Fields	Type	Description
text	String	Text which formating list refers to
formatting	List<Formating>	List of style modifications

Table 3.3: Fields' description of *FormattedText* class

Cell class

Cell class represents a unique table cell. To simplify, only *rowspan* and *colspan* attributes were considered because they have a lot of influence in table's structure. It also contains a *FormattedText* object, representing the actual text placed inside the cell.

Fields	Type	Description
rowspan	Integer	Number of rows the cell spans
colspan	Integer	Number of columns the cell spans
text	FormattedText	FormattedText object representing the text inside the text

Table 3.4: Fields' description of *Cell* class

TableFragment class

Remembering a small detail discussed in Section 3.2, some tables are composed by multiple other tables. *TableFragment* class was created to handle it, representing a small table (fragment) that is part of the whole table. Each *TableFragmemnt* object contains a list of header cells and a list of data cells. This cells are organized as lists of lists. The outer list represents a list of rows and the inner list represents a row: a list of cells.

Fields	Type	Description
headerCells	List<List<Cell>>	List of header rows. An header row is a list of cells
dataCells	List<List<Cell>>	List of data rows. A data row is a list of cells

Table 3.5: Fields' description of *TableFragment* class

Table class

Table class represents a table and all data related to it. It includes all fields located inside the *table-wrap* tag (in the XML article) and a list of paragraphs with a reference to the table.

Fields	Type	Description
pmcId	String	PMC article identification, "PMC" + a number (e.g. PMC4255783)
tabId	String	Table's identification
label	String	Table's label
title	String	Article's title
tableFragments	List<TableFragment>	List of *TableFragments* that composes the tabular data itself
caption	FormattedText	Table's caption
referencedTexts	List<FormattedText>	Set of paragraphs that have a reference to the table
footTexts	List<FormattedText>	Foot text divided in paragraphs

Table 3.6: Fields' description of *Table* class

3.3.2 PMC reader

PMC reader is responsible for finding tables inside PMC articles and transforming each one in a *Table* object. It uses *org.w3c.dom*[1], a Java package that provides useful methods to read XML and get the required information by representing each document as a tree of nodes.

First, the article metadata is read by finding some tags, namely the PubMed Central identifier (*article-id* tag) and title (*article-title* tag). Curiously, it was noticed that some articles do not have a title, so it was left empty in that cases.

Then, the reader will search for *table-wrap* tags and process them to extract the label, the caption, the footer, the tabId and the table itself. All *table* tags found inside *table-wrap* are transformed into *TableFragment* objects containing lists of header and data cells, according to the data model. Sometimes, *colspan* and *rowspan* attributes are not set, since they are only needed when the cell actually spans, that is when they are greater than 1. So, when these attributes are not found, they are set to 1. Table 3.7 associates tags found inside a *table-wrap* tag with their correspondent Java class or type that will represent it.

[1] `https://docs.oracle.com/javase/9/docs/api/org/w3c/dom/package-summary.html`

XML tag	XML parent tag	Java class/type
table-wrap	p or sec	Table
label	table-wrap	String
caption		FormattedTex
table-wrap-foot		List<FormattedText>
table		TableFragment
thead and tbody	table	List<List<Cell>>
tr	thead or tbody	List<Cell>
th and td	tr	Cell

Table 3.7: Java objects created by processing a *table-wrap* tag

After processing *table-wraps*, text that has references to tables is the only field that is missing. Figure 3.6 shows an example reference representation in XML and its correspondent online view. To find them, the reader searches all *xref* tags and filters them by *ref-type* attribute that must be equals to "table". It also gets *rid* attribute that identifies what table is being referenced. When a *xref* tag that references a table is found, the reader starts from its node and looks successively for parents until it finds a *p* node, that is the paragraph where the table is referenced. This is not computationally expensive, since the cycle will stop after the first iteration most of the cases. The reason to do it is that some *xref* tags are occasionally inside a formatting tag, so their first parent is not the desired *p* tag. Finally, by matching *rids* with *tabIds*, it is possible to assign extracted paragraphs with their correspondent tables.

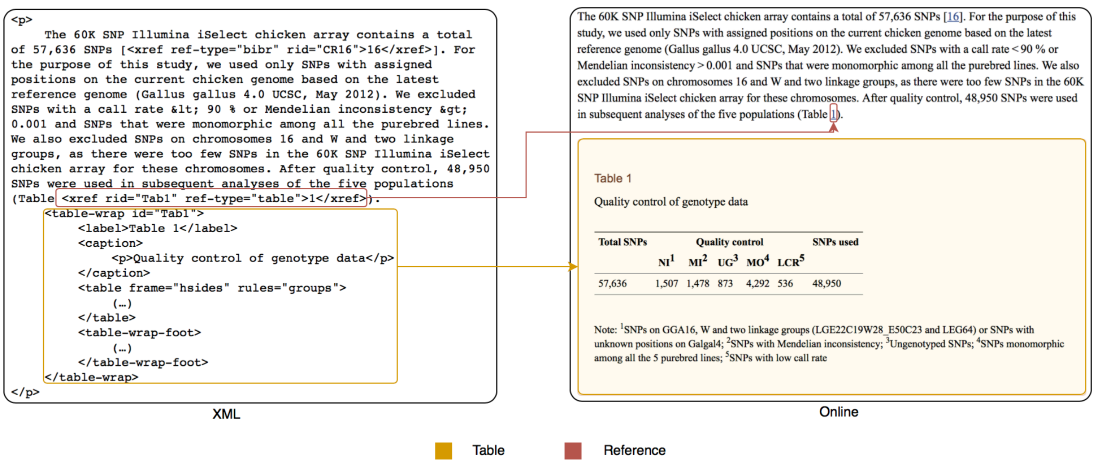

Figure 3.6: Example of a reference to a table (from PMC5002100)

All the fields that contain formatted text are processed in order to remove tags related to formatting. To do it, reader looks for tags inside the main node and creates a *Formatting* object for each one it founds. The actual text that is extracted obviously does not contain any tag. To simplify, just the more common formatting options are stored: italic, bold, sup, sub and ext-link (external link to a web site). Formulas were also stored as *Formatting* objects. All other tags that occasionally occur are ignored.

3.4 Neji

Named entity recognition task is assured by Neji framework through dictionary matching. A set of 51 dictionaries was used, containing biomedical concepts and their correspondent UMLS codes. They are divided in five categories: genes and proteins, anatomy, disorders, chemicals and biological processes.

Some obstacles were raised when integrating this framework. It was necessary to define what input give to Neji and also how to process the output in order to fit the data model.

3.4.1 Input

PMC parser stores tables as Java objects and Neji provides a Java API, but it is necessary to overcome some obstacles regarding the input formats:

- None of the input formats supported by Neji is directly applicable to the created data model;
- Neji was planned to run with files stored in disk, not data in memory.

Regarding the first problem, the most obvious way to adapt it is to run Neji for every text field, that is, to run it one time for each cell, caption, foot or paragraph with a reference. Unfortunately, this is quite inefficient. Every time Neji runs a pipeline it spends some time initializing all necessary data structures. For example, to annotate a single table with 4 columns and 5 rows, Neji would spend that initialization time more than 20 times. This would imply a big impact in performance, because a single initialization is supposed to cover one entire article, not just a few words.

Another option is to concatenate all fields, annotating one table at a time. But it raises another problem: how to distinguish caption's annotations from cell's annotations? Neji always splits the input in sentences, but paragraphs containing references can have more than one sentence, even cells can. The method found to fix this problem was to insert custom tags between fields and split Neji's output by them. These tags must be placed in single lines to guarantee that they do not mix with the "real" text. Listing 3.1 shows the text that results when this technique is applied to a table.

```
Populations
<!cell_delimiter!>
IGEAF
<!cell_delimiter!>
IPCL
<!cell_delimiter!>
Concordia
```

```
<!cell_delimiter!>
<!row_delimiter!>
<!header_data_delimiter!>
IPCL
<!cell_delimiter!>
0.0368*
<!cell_delimiter!>

<!cell_delimiter!>

<!cell_delimiter!>
<!row_delimiter!>
Concordia
<!cell_delimiter!>
0.0426*
<!cell_delimiter!>
0.0870*
<!cell_delimiter!>

<!cell_delimiter!>
<!row_delimiter!>
Puerto Yeruá
<!cell_delimiter!>
0.0671*
<!cell_delimiter!>
0.1309*
<!cell_delimiter!>
0.0261*
<!cell_delimiter!>
<!row_delimiter!>
<!header_data_delimiter!>
<!fragment_delimiter!>
<!table_field_delimiter!>
Pairwise FSTvalues (Weir and Cockerham 1984) for the four A. fraterculus populations analyzed.
<!table_field_delimiter!>
*Statistical significance P < 0.05
<!paragraph_delimiter!>
<!table_field_delimiter!>
Analysis of genotypic frequencies across all loci for each pair-wise comparison (G Test, Fisher´s method) showed significant differences between all pairs of populations (P<0.05; see details of allelic and genotypic frequencies for each locus in
```

```
↪ each population in Table S1 and S2 in additional files 3 and 4,
↪ respectively). Pair-wise FST values significantly differed from
↪ zero (Table 1). The highest level of genetic differentiation (
↪ FST = 0.1309) was observed between Puerto Yeruá and IPLC,
↪ whereas the lowest value of differentiation (FST= 0.0261) was
↪ observed between Concordia and Puerto Yeruá. FIS values showed
↪ low incidence of inbreeding in all populations (Concordia FIS =
↪ 0.13; IPCL FIS = 0.25; IGEAF FIS = 0.16; Puerto Yeruá FIS =
↪ 0.21).
<!paragraph_delimiter!>
<!table_field_delimiter!>
```

Listing 3.1: Example Neji's input table (from PMC4255783)

There is only a small con, a collision may occur if a tag is equals to a sentence of the "real" text. The system is prepared to detect this kind of error, by ensuring that the number of items forwarded to Neji is equals to the number of items found in its output.

To minimize even more the impact of Neji's initialization time, several tables can be joined based on the same principle. Therefore, tables are concatenated and forwarded to Neji in chunks. Table 3.8 summarizes all custom tags used when building Neji's input.

Custom tag	Description
<!table_delimiter!>	Separates tables.
<!table_field_delimiter!>	Separates table fields: list of table fragments, caption, list of paragraphs from foot and list of paragraphs with a reference to a table
<!cell_delimiter!>	Separates cells
<!row_delimiter!>	Separates rows
<!header_data_delimiter!>	Separates header cells from data cells
<!fragment_delimiter!>	Separates table fragments
<!paragraph_delimiter!>	Separates paragraphs from foot or with a reference to a table

Table 3.8: Tags used in Neji's input

Remembering the second problem, Neji was made to read files in disk. As it is file oriented, it has techniques to process large series of files as fast as possible. It can load a set of files, usually in the same directory, and annotate them all while sharing some resources, like dictionaries and text parsers. Multithreading is also supported and can be used to improve even more the performance. But, storing tables on disk and read them right after it would not be worth. To adapt it, it was necessary to create two new classes, *InputTable* and *TableBatchExecutor*, in order to replace the file oriented ones, *InputFile* and *FileBatchExecutor*. Neji's code was used as a base of these classes, so only some changes were made.

InputTable represents a set of tables to be annotated by Neji at once. Instead of returning the content of a local file like *InputFile* class does, this class returns the input discussed above. So, it does all the necessary concatenations to build a unique input string and afterwards it also splits Neji's output and assign annotations with their correspondent fields.

TableBatchExecutor receives a list of *Table* objects and splits them into smaller lists according to the number of threads used. It creates *InputTable* objects to be further processed by Neji. For example if Neji is runnig with 4 threads and *FileBatchExecutor* receives 1000 tables, it will create 4 *InputFile* objects, each one with approximately 250 tables.

3.4.2 Output

Neji can return its results as several distinct formats: BioC, A1, CoNLL, JSON and BC2. JSON was the chosen one, because it returns all produced information (some formats omit some data) and at the same time it is easy to process in Java by using the *gson*[2] library. Considering the JSON output, processed text is divided in sentences and for each one there are its start and end positions, its text and all annotations found. Listing 3.2 shows the JSON output produced by Neji containing an annotated sentence (Table 1 from PMC4255783).

```
{
  "id": 44,
  "start": 1195,
  "end": 1405,
  "text": "The highest level of genetic differentiation (FST = 0.1309) was observed between Puerto Yeruá and IPLC, whereas the lowest value of differentiation (FST= 0.0261) was observed between Concordia and Puerto Yeruá.",
  "terms": [
    {
      "ids": "UMLS:C0917892:T045:PROC",
      "score": 1,
      "id": 1,
      "start": 1216,
      "end": 1239,
      "text": "genetic differentiation",
      "terms": []
    },
    {
      "ids": "UMLS:C1414830:T116:PRGE",
      "score": 1,
```

[2] https://github.com/google/gson

```
      "id": 2,
      "start": 1241,
      "end": 1244,
      "text": "FST",
      "terms": []
    }
  ]
}
```

Listing 3.2: Example Neji's JSON output

Each annotation also contains its start and end positions, its UMLS codes, its text and another annotations that it may contain. Remember that submitted inputs are composed by a set of tables, each one containing a set of distinct fields. Because of that, start and end positions will be based on the whole input, so it is necessary to fix them to the correct positions. This is made by subtracting the start position of the first sentence of each field to all remaining positions.

Because of using text processing methods, Neji removes white spaces that may exist at the beginning of each sentence. It happens that some table cells starts with white spaces for indentation purposes. To avoid to lose this data, start and end positions are also fixed by adding the number of white spaces that exists at the beginning of the original sentence to the positions of annotations.

For each annotation found, an *Annotation* object (containing the start position, the text and UMLS codes) was created and assigned to its correspondent *FormattedText* object.

3.5 Elasticsearch

Elasticsearch is used to index gathered data and to make queries in order to retrieve tables. So, it is important to discuss some details regarding indexing and querying operations.

3.5.1 Indexing

To index stored tables according to the data model, it is necessary to represent them in JSON. So, a mapping was created to define the JSON structure used to index tables.

```
{
  "properties" : {
    "table" : {
      "properties" : {
        "header" :{
          "properties": {
            "text" : { "type" : "text"},
            "rowspan" : {"type" : "integer", "index" : "false"},
```

```
            "colspan" : {"type" : "integer", "index" : "false"},
            "annotations": {
              "type" : "nested",
              "properties": {
                "text" : {"type" : "text"},
                "ids" : {"type" : "text"},
                "start" : {"type" : "integer", "index" : "false"}
              }
            },
            "formatting": {
              "properties": {
                "text" : {"type" : "text", "index" : "false"},
                "tag" : {"type" : "text", "index" : "false"},
                "start" : {"type" : "integer", "index" : "false"}
              }
            }
          }
        },
        "data" :{
          "properties": {
            "text" : {"type" : "text"},
            "rowspan" : {"type" : "integer", "index" : "false"},
            "colspan" : {"type" : "integer", "index" : "false"},
            "annotations": { (...) },
            "formatting": { (...) }
          }
        }
      }
    },
    "caption": {
      "properties": {
        "text" : {"type" : "text"},
        "annotations": { (...) },
        "formatting": { (...) }
      }
    },
    "foot": {
      "properties": {
        "text" : {"type" : "text"},
        "annotations": { (...) },
        "formatting": { (...) }
      }
    },
```

```
    "text": {
      "properties" :{
        "text" : {"type" : "text"},
        "annotations":{ (...) },
        "formatting": { (...) }
      }
    },
    "pmcId": {"type" : "integer"},
    "tabId": {"type" : "text"},
    "pmcTitle": {"type" : "text"},
    "label": {"type" : "text"}
  }
}
```

Listing 3.3: Elasticseach mapping

Listing 3.3 contains that mapping (note that *annotations* and *formatting* properties are omitted when repeated). Basically, each document represents a *Table* object from data model and results from transforming each Java field in a key and a value, including inner objects. Elasticsearch does not provide an explicit array type, because all fields can be used as lists containing several values. Arrays of arrays are also available.

Each table is composed by eight main fields: *table*, *caption*, *foot*, *text*, *pmcId*, *tabId*, *pmcTitle* and *label*. While the last four fields are simple types (text or integer), the other four are more complex because they contain inner fields, so they will be more detailed.

Table represents a list of *TableFragment*, containing two fields: *header* and *data*. These fields are lists of lists, that is lists of rows, being each row a list of cells. A cell is represented by its *text*, *rowspan*, *colspan*, *annotations* and *formatting*.

Caption, ***foot*** and ***text*** contain the same fields: *text*, *annotations* and *formatting*. While *caption* has just a value, the other two are lists, because they may be composed by multiple paragraphs.

Annotations field represents a list of annotations. Each one contains the *text*, *start* position and a list of UMLS codes (*ids*). This field is the only one that is set as *nested*, meaning that Elasticsearch will maintain a correspondence between the fields of the same object when indexing a list. In other words, when having more than one object inside *annotations*, it is possible to distinguish which *text* corresponds to which *ids*. It would not be possible to do, if the type were the default (*object*), because in this case Elasticsearch would just index *text*, *ids* and *start* as independent arrays, not granting object's order.

Note that some fields have "index" value as "false". This means that they are not indexed by Elasticsearch, so it is not possible to make queries based on them. It was set on fields that store information about visualization, like start positions of annotations, colspan, rowspan and all formatting data.

3.5.2 Querying

A general query was implemented in order to retrieve tables based on text contained in caption, foot, text with an reference to a table and the table itself. All fields are queried by the operator *match* that performs a full-text search. To influence the weights, *boost* was used to multiply all fields' weights by a factor. So, *header* cells and *caption* have the maximum boost (5), *data* cells and *text* have a medium one (3) and *foot* has no boost (1). Listing 3.4 shows an example query for the word “heart”.

```
{
   "query": {
      "bool": {
         "should": [
            { "match": {
                  "table.header.text": {
                     "query": "heart",
                     "boost": 5
                  }
               }
            },
            { "match": {
                  "table.data.text": {
                     "query": "heart",
                     "boost": 3
                  }
               }
            },
            { "match": {
                  "caption.text": {
                     "query": "heart",
                     "boost": 5
                  }
               }
            },
            { "match": {
                  "foot.text": {
                     "query": "heart",
                     "boost": 1
                  }
               }
            },
            { "match": {
                  "text.text": {
                     "query": "heart",
```

```json
              "boost": 3
            }
          }
        }
      ]
    }
  }
}
```

Listing 3.4: Example JSON of a general query (query = “heart”)

Chapter 4

Results and Web interface

This chapter presents results obtained by testing the whole system with distinct parameters or inputs. This was made to evaluate the performance of some modules and to validate the solution. Additionally, the Web interface is presented and illustrated by figures of its views. The main operations performed by the back-end are described too.

4.1 Results

Some tests were run to validate the solution and to evaluate performance:

- The Neji module was tested in order to understand the impact of annotating chunks of tables in terms of time;
- The PMC Parser module was singly tested with a larger dataset to guarantee that there is no parsing errors;
- The whole system was tested to prove its capabilities: it can read, annotate and retrieve tables.

Table 4.1 contains information about the machine that was used to run all tests. This machine was always running the Web interface and Elasticsearch servers.

CPU	2,6 GHz Intel Core i7 quad-core
RAM	16 GB 2133 MHz LPDDR3
Disk	256 GB SSD
OS	macOS High Sierra

Table 4.1: Machine used to run tests

Neji

Each time Neji runs, it processes a set of tables represented by a unique string. This strategy was adopted to minimize Neji's initialization time. To understand the effectiveness of such technique, the graphic of Figure 4.1 shows the time spent by Neji when annotating chunks of tables with distinct sizes. These results are referring to tests that were made running Neji with 4 threads to annotate 2652 tables from 1000 files. More detailed results are available in Table 4.2.

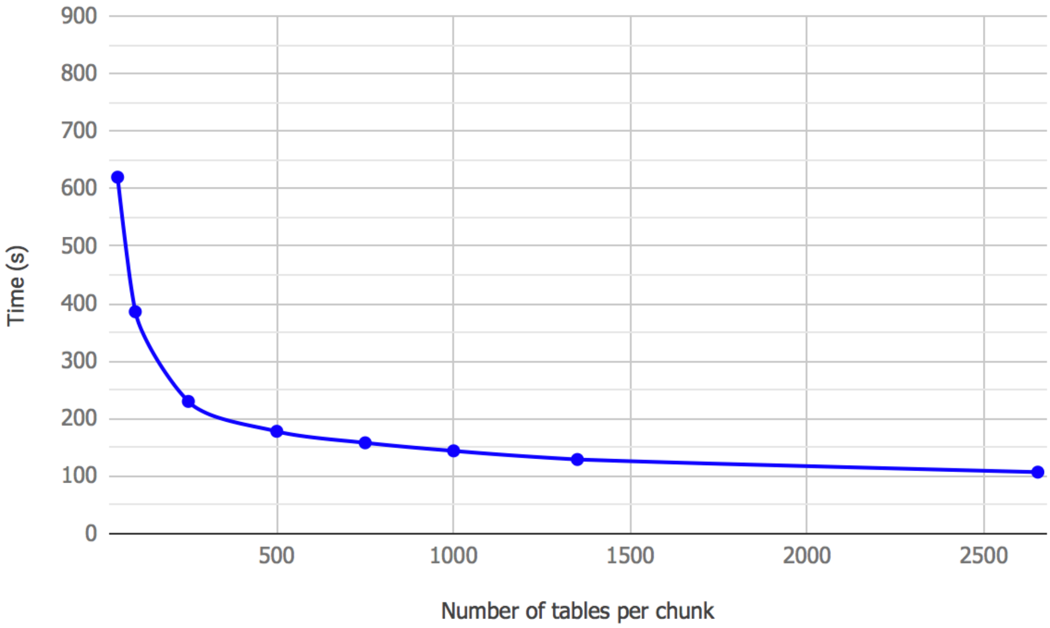

Figure 4.1: Time spent when running Neji with different different number of tables per chunk

Chunks		Time		
Size	**Number**	**Parse**	**Annotations**	**Index**
50	54	7 s	10 min 20 s	11 s
100	27	7 s	6 min 26 s	11 s
250	11	7 s	3 min 50 s	11 s
500	6	7 s	2 min 58 s	11 s
750	4	7 s	2 min 38 s	11 s
1000	3	7 s	2 min 24 s	11 s
1350	2	7 s	2 min 10 s	11 s
2652	1	7 s	1 min 47 s	11 s

Table 4.2: Results from running Neji with different number of tables per chunk

Small chunks of tables have a huge impact in performance. This proves that it would be impracticable to run Neji module one time for each table field. Grouping large quantities of tables drastically improved performance. However, this improvement is less significant when comparing the biggest chunks. Thus, chunk size must be high to obtain a good performance but without exaggerating and taking the risk of making the machine run out of memory.

As it was expected, using different sizes for chunks does not influence the time spent on parsing and indexing operations.

PMC Parser

To test the robustness of the PMC Parser, 200 thousand articles were read. It successfully located and parsed 593 199 tables and no errors were found. The entire process took 26 minutes and 14 seconds. Some collected statistics are available in Table 4.3.

Number of tables	**total**	593 199 (≈ 3 tables/article)
	with multiple fragments	660 (≈ 0,1 %)
	with no header	201 675 (≈ 34 %)
	not referenced in text	209 484 (≈ 35 %)
Total number of cells		37 203 636 (≈ 63 cells/table)
Number of articles that contain tables		153 050 (≈ 77 %)

Table 4.3: Statistics about parsed tables

Biomedical articles usually contain many tables and these tables are big, since there are approximately 3 tables per article and the average number of cells per table is 63. However, about 23% of the articles do not contain tables. Figure 4.2 shows the number of tables contained by the parsed articles.

Tables with no header are common (34%). Note that headers are identified in the original XML files but it is possible that some tables are wrongly marked. If this is the case, function analysis operations could be used in order to extract possible headers when they are not identified.

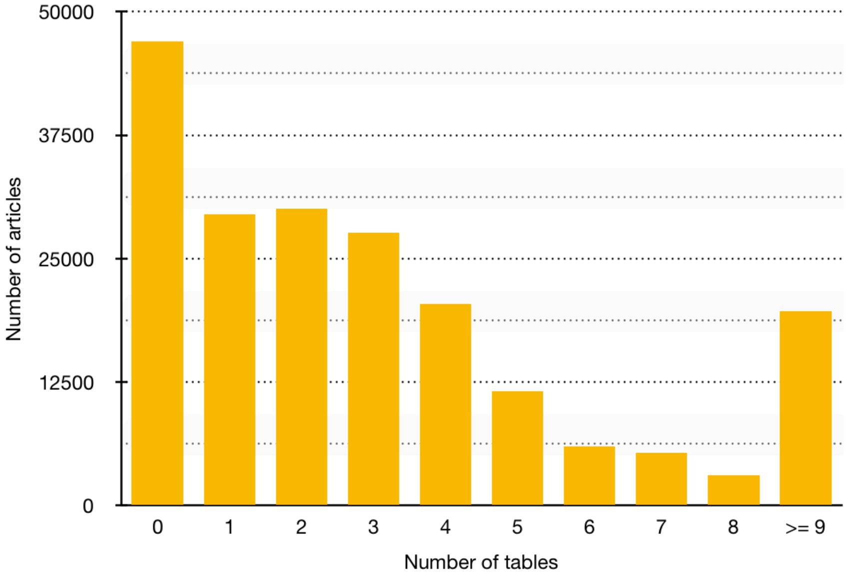

Figure 4.2: Number of tables per article

There are a significant percentage of tables (35%) that are not referenced in text, so they will be retrieved with less context. Ten articles containing these kind of tables were manually inspected to understand if they are not referenced in text. Most of them actually contain tables that are not referenced in text. However, two of these articles (PMC3113366 and PMC3560439) revealed that there are tables composed by multiple fragments that were parsed as independent tables. The problem is that the existing references are pointing to the group and not to individual fragments.

By analyzing the correspondent XML files, it was found that it happened because these tables with multiple fragments are stored in a different way than the ones that were considered. In these cases, multiple fragments are delimited by the same tag (*table-wrap*) as normal tables are and they are placed inside another tag indicating that a set of tables is a unique table. Curiously, the tags used in both inspected articles are not the same, as showed in Figure 4.3.

The number of parsed tables composed by multiple other tables is very small, only 0,1%. However, it was found that there are more tables with this characteristic, as stated before. A simple method was written in order to count the number of tables that follow the two found patterns illustrated in Figure 4.3 and it only matched 1 360 tables. The impact of these cases is not high, since the number of parsed tables not referenced in text is greater than 200 thousand.

```
<table-wrap-group id="tab1" position="float">
    <label>
        Table 1
    </label>
    <caption>
        <p>Prevalence of HPV at different anatomical sites for circumcised
            and uncircumcised men in various studies in different countries.
        </p>
    </caption>
    <table-wrap id="tab1a" position="anchor">
        (…)
    </table-wrap>
    <table-wrap id="tab1b" position="anchor">
        (…)
    </table-wrap>
    (…)
</table-wrap-group>
```

Table 1 from PMC3113366

```
<table-wrap id="T1" orientation="portrait" position="float">
    <label>
        Table 1
    </label>
    <caption>
        <title>Gene expression changes common to aging and each stress</title>
    </caption>
    <list list-type="alpha-lower">
        <list-item>
            <p>18 genes up-regulated in aging and all other stresses</p>
            <p>
                <table-wrap orientation="portrait" id="d35e208" position="float">
                    (…)
                </table-wrap>
            </p>
        </list-item>
        <list-item>
            (…)
        </list-item>
    </list>
</table-wrap>
```

Table 1 from PMC3560439

Figure 4.3: Tags used to define multiple fragments

System

The entire system was tested with 10 thousand files and the chunk size was set to 2 000. This process took 13 minutes and 20 seconds and indexed 14 911 tables. The Neji module was the most expensive one, since it took about 10 times more than the other modules. In terms of disk size, the original files take 563.3 MB and the final index was reduced to 89.2 MB. Although a lot of text of the original articles is discarded, multiple annotations are extracted and they will make the index grow in size. All these presented results are available and compacted in table 4.4.

Number of	**Articles**	10 000
	Tables	14 911
Size	**Index**	89.2 MB
	Files	563.3 MB
Time	**PMC parsing**	1 min 2 s
	Neji	11 min 18 s
	Elasticsearch	1 min
	Total	13 min 20 s

Table 4.4: Results from testing the entire system

4.2 Web interface

A web interface was developed in order to allow end-users to query the indexed data and to visualize results. The last test presented in Section 4.1 provided indexed tables that were used to feed the web interface.

4.2.1 Overview

The web interface was developed in Django [51], a free and open source high-level Python Web framework. It allows users to query indexed tables stored in the Elasticsearch index and to navigate through the results visualizing them. Its pipeline of operations can be described as:

- reads a query submitted by the user;
- constructs a JSON string that represents the query in DSL;
- sends a request through the Elasticsearch API for Python;
- reconstructs retrieved tables, highlights annotated concepts and formats text;
- shows results to the user.

Figure 4.4 shows the page of results to a query with the text "heart". All results are presented by score (already sorted by Elasticsearch). At the top, each item contains the article's title and the table's label with hyperlinks to the original article and table respectively. These links are build using *pmcId* and *tabId*, always by following the same structures:

Link to an article

https://www.ncbi.nlm.nih.gov/pmc/articles/**a**, where **a** is the PMC identification.

Link to an article's table

https://www.ncbi.nlm.nih.gov/pmc/articles/**a**/table/**b**, where **a** is the PMC identification and **b** is the *tabId*.

The content of each result is divided in two parts. On the right, all paragraphs that contain a reference to the table are presented. On the left, the table itself is presented containing the caption above and the foot below it.

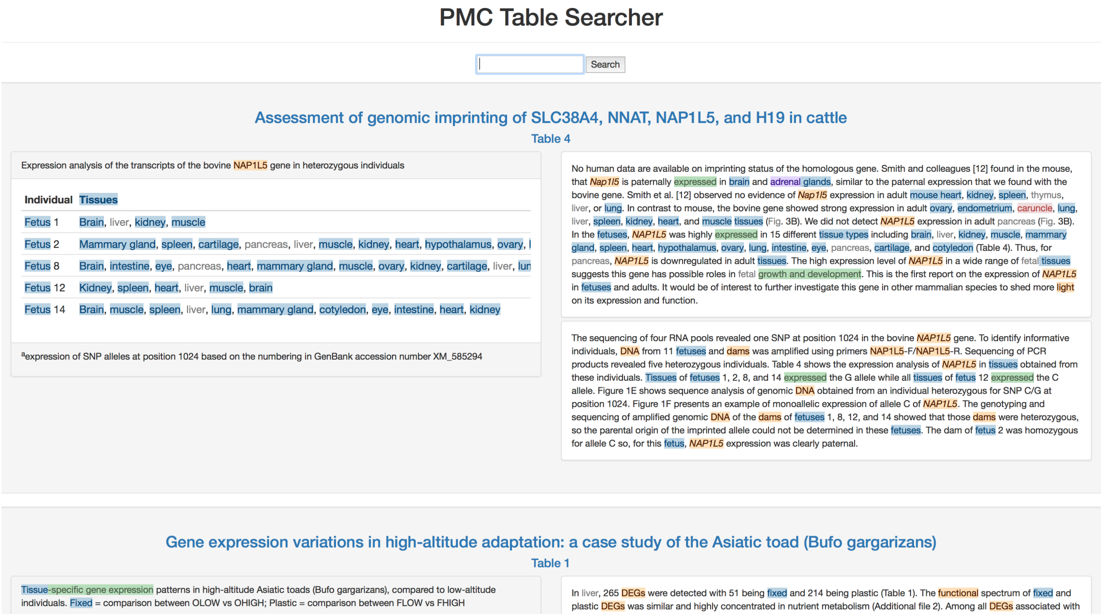

Figure 4.4: Web interface showing results (query = "heart")

Elasticsearch can return a specific number of results from a start point. So, results are showed in chunks of 10 and the next set of results are only requested when the user changes the page.

4.2.2 Reconstructing tables

In order to show tables to the user, it is necessary to transform the JSON originally sent to Elasticsearch during indexing time into an HTML representation.

Listing 4.1 has a table fragment JSON example. To simplify, some rows were removed and all cell fields were omitted except the text one.

```
{
  "header": [
    [
      {
        "text": "Individual"
      },
      {
        "text": "Tissue"
      },
      {
        "text": "Alleles expressed at position 9188a"
      }
    ]
```

```
  ],
  "data": [
     [
        {
           "text": "Fetus 2"
        },
        {
           "text": "Kidney, spleen, cartilage, pancreas"
        },
        {
           "text": "T/G"
        }
     ],
     [
        {
           "text": "Fetus 4"
        },
        {
           "text": "Kidney, heart, brain, lung"
        },
        {
           "text": "T/G"
        }
     ],
     (...)
  ]
}
```

Listing 4.1: Example table JSON (simplified for clarity)

Header and data fields contain arrays of rows, where a row is an array of cells. With this information, to reconstruct tables is trivial. *Theader* and *tbody* tags are created containing the respective cells. For each row, a *tr* tag is also created with a list of cells. For header cells, *th* tag is used while data cells are stored as *td* tags. *Rowspan* and *colspan* fields were also added to the HTML representation in order to mantain the original structure of tables.

4.2.3 Annotations and text formatting

To make annotations perceptible, text is highlighted with a different color according to its category. The last field of the UMLS code defines all categories. When an annotation has codes with more than one category, it is set as "Ambiguous", once there is no way to know what is the right one.

When the mouse is hover an annotation, a popover appears showing its category and the list of UMLS codes converted to a small description, as illustrated in Figure 4.5. These descriptions are stored in a local file that is loaded when the server starts. It is transformed in a dictionary where the key is the UMLS id and the value is its description/name.

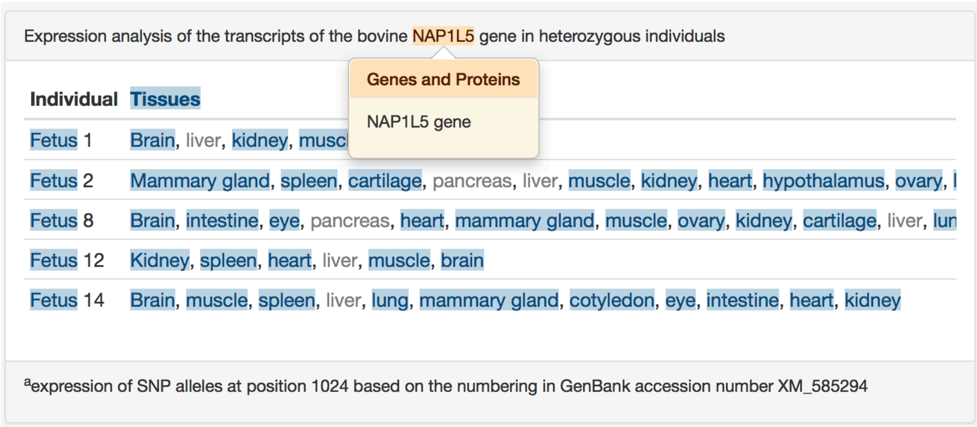

Figure 4.5: Annotation popover

Highlights are based on *span* tags that are set with different CSS classes according to the category. To insert these tags in the original text, all annotations are sorted by its start position. Next, a new string is built by iterating the original text character by character. When it founds an annotation starting or ending position, the *span* tag containing the respective class is added to the string .

Formatted text is constructed by the same way and at the same time as annotations, but by inserting HTML proper tags. Only formulas stored as Mathematical Markup Language are interpreted by the web interface, since this language is supported by HTML. LaTeX formulas are ignored. Table 4.5 shows a comparison between tags used by PMC in their XML articles and their correspondent HTML tag used in the web interface.

PMC XML tag	HTML tag
<italic>	<i>
<bold>	<b>
<sup>	<sup>
<sub>	<sub>
<ext-link>	<a>
<mml:math>	<math>

Table 4.5: Tags used to format text: XML vs HTML

Chapter 5

Conclusion

The main objective of this dissertation was to develop an information retrieval and information extraction system focused on tabular data from biomedical articles. To better understand the problem, information retrieval and information extraction topics were studied and discussed, including some frameworks that implements them. Even thought there are very robust and well-know ways to apply these concepts to plain text, tabular data raises some additional obstacles and there is much less research regarding it.

The objectives have been fulfilled. A system was implemented to annotate and index tables extracted from the PMC collection. A web interface was also developed to allow users to query the index and visualize annotated tables that are retrieved. This can be considered a contribute to the biomedical community because it provides a different way to retrieve information from articles. It obviously does not replace traditional systems that retrieve articles, but could be used as a complement, an additional feature that allows researchers to search information stored in tables.

However, this work does not close the topic about table mining. It can be used as a starting point for further research, being opened to receive updates that can enrich the solution. Some possible improvements are:

- **Add support to more file formats**: table extraction is only performed over XML files. Although, there are another formats (e.g. PDF or TXT) that could be also considered by creating specific readers for them.

- **Include tables from another sources**: PMC Open Access Subset was the dataset used, so some details related to it (e.g. *pmcId*) was introduced to make the most of the available information. Although, it can be easily adapted to carry tables from distinct sources.

- **Implement other information extraction methods**: only named entity recognition was performed. There are other possible techniques regarding information extraction that could be implemented (e.g. extract relations between cells).

- **Improve named entity recognition**: the solution found to perform concept recognition works by adding some tags to the text to be annotated. This means that Neji will waste some time trying to annotate tags. Some research can be made in order to find a way to avoid it and eventually improve performance.

- **Integrate data curation**: automatic annotations are not perfect. To improve the quality of annotations, a data curation option could be integrated in order to allow area experts to manually remove, add and validate annotations.

Bibliography

[1] U.S. National Library of Medicine. Citations Added to MEDLINE by Fiscal Year. [Online]. Available: `https://www.nlm.nih.gov/bsd/stats/cit_added.html`. [Accessed: 10-11-2017].

[2] Aaron M Cohen and William R Hersh. A survey of current work in biomedical text mining. *Briefings in bioinformatics*, 6(1):57–71, 2005.

[3] Jing Fang, Prasenjit Mitra, Zhi Tang, and C Lee Giles. Table header detection and classification. In *AAAI*, pages 599–605, 2012.

[4] Vidhya Govindaraju, Ce Zhang, and Christopher Ré. Understanding tables in context using standard nlp toolkits. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 658–664, 2013.

[5] Hong-Jie Dai, Yen-Ching Chang, Richard Tzong-Han Tsai, and Wen-Lian Hsu. New challenges for biological text-mining in the next decade. *Journal of computer science and technology*, 25(1):169–179, 2010.

[6] Carmen De Maio, Giuseppe Fenza, Vincenzo Loia, and Mimmo Parente. Text mining basics in bioinformatics. 2016.

[7] Marti Hearst. What is text mining. *SIMS, UC Berkeley*, 2003.

[8] Elizabeth D Liddy. Natural language processing. 2001.

[9] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[10] Sunita Sarawagi et al. Information extraction. *Foundations and Trends® in Databases*, 1(3):261–377, 2008.

[11] Lars Juhl Jensen, Jasmin Saric, and Peer Bork. Literature mining for the biologist: from information retrieval to biological discovery. *Nature reviews genetics*, 7(2):119, 2006.

[12] Stéphane M Meystre, Guergana K Savova, Karin C Kipper-Schuler, John F Hurdle, et al. Extracting information from textual documents in the electronic health record: a review of recent research. *Yearb Med Inform*, 35(8):128–144, 2008.

[13] Stefan Büttcher, Charles LA Clarke, and Gordon V Cormack. *Information retrieval: Implementing and evaluating search engines*. Mit Press, 2016.

[14] Raymond J Mooney. CS 371R: Information Retrieval and Web Search. [Online]. Available: `https://www.cs.utexas.edu/users/mooney/ir-course/`. [Accessed: 15-04-2018].

[15] Dan Jurafsky and James H Martin. *Speech and language processing*, volume 3. Pearson London:, 2014.

[16] David Campos, Sérgio Matos, and José Luís Oliveira. Current methodologies for biomedical named entity recognition. *Biological Knowledge Discovery Handbook: Preprocessing, Mining, and Postprocessing of Biological Data*, pages 839–868, 2013.

[17] Allen C Browne, Alexa T McCray, and Suresh Srinivasan. The specialist lexicon. *National Library of Medicine Technical Reports*, pages 18–21, 2000.

[18] Kenji Sagae and Jun'ichi Tsujii. Dependency parsing and domain adaptation with lr models and parser ensembles. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 2007.

[19] Alias-i. LingPipe. [Online]. Available: `http://alias-i.com/lingpipe/`. [Accessed: 11-06-2018].

[20] The Apache Software Foundation. Apache OpenNLP. [Online]. Available: `http://opennlp.apache.org/`. [Accessed: 11-06-2018].

[21] Ken-ichiro Fukuda, Tatsuhiko Tsunoda, Ayuchi Tamura, Toshihisa Takagi, et al. Toward information extraction: identifying protein names from biological papers. In *Pac symp biocomput*, volume 707, pages 707–718, 1998.

[22] U.S. National Library of Medicine. UMLS - Metathesaurus Release Statistics. [Online]. Available: `https://www.nlm.nih.gov/research/umls/knowledge_sources/metathesaurus/release/statistics.html`. [Accessed: 11-06-2018].

[23] Matthew Francis Hurst. The interpretation of tables in texts. 2000.

[24] Matthew Hurst. Layout and language: Challenges for table understanding on the web. In *Proceedings of the International Workshop on Web Document Analysis*, pages 27–30, 2001.

[25] W3C Recommendations. Tables in HTML documents. [Online]. Available: https://www.w3.org/TR/html401/struct/tables.html. [Accessed: 28-11-2017].

[26] Yalin Wang and Jianying Hu. Detecting tables in html documents. In *International Workshop on Document Analysis Systems*, pages 249–260. Springer, 2002.

[27] Hsin-Hsi Chen, Shih-Chung Tsai, and Jin-He Tsai. Mining tables from large scale html texts. In *Proceedings of the 18th conference on Computational linguistics-Volume 1*, pages 166–172. Association for Computational Linguistics, 2000.

[28] Florence Folake Babatunde, Bolanle Adefowoke Ojokoh, and Samuel Adebayo Oluwadare. Automatic table recognition and extraction from heterogeneous documents. *Journal of Computer and Communications*, 3(12):100, 2015.

[29] Yalin Wang and Jianying Hu. A machine learning based approach for table detection on the web. In *Proceedings of the 11th international conference on World Wide Web*, pages 242–250. ACM, 2002.

[30] Kristina Lerman, Lise Getoor, Steven Minton, and Craig Knoblock. Using the structure of web sites for automatic segmentation of tables. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 119–130. ACM, 2004.

[31] Bernhard Krüpl and Marcus Herzog. Visually guided bottom-up table detection and segmentation in web documents. In *Proceedings of the 15th international conference on World Wide Web*, pages 933–934. ACM, 2006.

[32] Burcu Yildiz, Katharina Kaiser, and Silvia Miksch. pdf2table: A method to extract table information from pdf files. In *IICAI*, pages 1773–1785, 2005.

[33] PDFTOHTML conversion program. [Online]. Available: http://pdftohtml.sourceforge.net/. [Accessed: 23-12-2017].

[34] Ermelinda Oro and Massimo Ruffolo. Trex: An approach for recognizing and extracting tables from pdf documents. In *Document Analysis and Recognition, 2009. ICDAR'09. 10th International Conference on*, pages 906–910. IEEE, 2009.

[35] Ying Liu, Prasenjit Mitra, and C Lee Giles. Identifying table boundaries in digital documents via sparse line detection. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 1311–1320. ACM, 2008.

[36] Elasticsearch: RESTful, Distributed Search & Analytics | Elastic. [Online]. Available: https://www.elastic.co/products/elasticsearch. [Accessed: 19-04-2018].

[37] The Apache Software Foundation. Apache Solr. [Online]. Available: https://lucene.apache.org/solr/. [Accessed: 14-06-2018].

[38] Rafał Kuć and Marek Rogoziński. *Mastering Elasticsearch*. Packt Publishing Ltd, 2015.

[39] Historical trend of search engines popularity. [Online]. Available: https://db-engines.com/en/ranking_trend/search+engine, note = [Accessed: 19-04-2018].

[40] Use Cases - Elastic Stack Success Stories. [Online]. Available: https://www.elastic.co/use-cases. [Accessed: 19-04-2018].

[41] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. " O'Reilly Media, Inc.", 2015.

[42] Mapping | Elasticsearch Reference [6.2] | Elastic. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html. [Accessed: 27-04-2018].

[43] Query DSL | Elasticsearch Reference [6.2] | Elastic. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html. [Accessed: 27-04-2018].

[44] Alan R Aronson and François-Michel Lang. An overview of metamap: historical perspective and recent advances. *Journal of the American Medical Informatics Association*, 17(3):229–236, 2010.

[45] Michael A Tanenblatt, Anni Coden, and Igor L Sominsky. The conceptmapper approach to named entity recognition. In *LREC*, pages 546–51, 2010.

[46] Robert Leaman and Graciela Gonzalez. Banner: an executable survey of advances in biomedical named entity recognition. In *Biocomputing 2008*, pages 652–663. World Scientific, 2008.

[47] David Campos, Sérgio Matos, and José Luís Oliveira. A modular framework for biomedical concept recognition. *BMC bioinformatics*, 14(1):281, 2013.

[48] Andrew Kachites McCallum. MALLET: A Machine Learning for Language Toolkit. [Online]. Available: http://mallet.cs.umass.edu. [Accessed: 11-06-2018].

[49] Xiaohua Zhou, Xiaodan Zhang, and Xiaohua Hu. Dragon toolkit: incorporating auto-learned semantic knowledge into large-scale text retrieval and mining. In *Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE International Conference on*, volume 2, pages 197–201. IEEE, 2007.

[50] PMC - NCBI. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/. [Accessed: 26-05-2018].

[51] Django Software Foundation. The Web framework for perfectionists with deadlines | Django. [Online]. Available: https://www.djangoproject.com. [Accessed: 18-06-2018].